
A New Approach to Probabilistic Programming Inference

Frank Wood

Department of Engineering
University of Oxford

Jan Willem van de Meent

Department of Statistics
Columbia University

Vikash Mansinghka

Computer Science & AI Lab
Massachusetts Institute of Technology

Abstract

We introduce and demonstrate a new approach to inference in expressive probabilistic programming languages based on particle Markov chain Monte Carlo. Our approach is simple to implement and easy to parallelize. It applies to Turing-complete probabilistic programming languages and supports accurate inference in models that make use of complex control flow, including stochastic recursion. It also includes primitives from Bayesian nonparametric statistics. Our experiments show that this approach can be more efficient than previously introduced single-site Metropolis-Hastings methods.

1 Introduction

Probabilistic programming differs substantially from traditional programming. In particular, probabilistic programs are written with parts not fixed in advance that instead take values generated at runtime by random sampling procedures. Inference in probabilistic programming characterizes the conditional distribution of such variables given observed data assumed to have been generated by executing the probabilistic program. Exploring the joint distribution over program execution traces that could have generated the observed data using Markov chain sampling techniques is one way to produce such a characterization.

We propose a novel combination of ideas from probabilistic programming [3] and particle Markov chain Monte Carlo (PMCMC) [1] that yields a new scheme for exploring and characterizing the set of probable execution traces. As our approach is based on repeated simulation of the probabilistic program, it is easy to implement and parallelize. We show that our approach

supports accurate inference in models that make use of complex control flow, including stochastic recursion, as well as primitives from nonparametric Bayesian statistics. Our experiments also show that this approach can be more efficient than previously introduced single-site Metropolis-Hastings (MH) samplers [12].

2 Language

The probabilistic programming system Anglican¹ exists in two versions. The results in this paper were obtained using what is now called *interpreted* Anglican² which employed an interpreted execution model and a language syntax derived from the Venture³ modeling language. Since the time of original publication, both the syntax and execution model have been changed. The Venture style syntax is now deprecated, with the new version using a syntax much closer to that of Anglican's host language Clojure. This new Anglican version⁴, simply called Anglican, is a compiled language; a continuation-passing style (CPS) transformation compiles Anglican programs into Clojure programs that are then subsequently compiled to Java Virtual Machine (JVM) bytecode by the Clojure just-in-time compiler.

The change in language execution model and syntax affect neither the substance or the claims of this paper, however, readers wishing to experiment with the language starting from the code examples appearing in Section 4 would do well to note this evolution. What does change is the absolute time required to perform inference in the given models. In general, the latest compiled version of Anglican is ten to one hundred times faster than its interpreted ancestor.

This paper, which originally appeared in the Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS) 2014, has been updated to reflect changes in the latest version of the Anglican language.

¹<http://www.robots.ox.ac.uk/~fwood/anglican/>

²<http://bitbucket.org/probprog/interpreted-anglican>

³<http://probcomp.csail.mit.edu/venture/>

⁴<http://bitbucket.org/probprog/anglican>

2.1 Original Venture-Style Syntax

The original, deprecated Anglican syntax is a Scheme/Lisp-dialect that is extended with three top-level special forms that we refer to as directives

```
|| [assume symbol <expr>]
|| [observe <expr> <const>]
|| [predict <expr>]
```

Here, each `<expr>` is a Scheme/Lisp-syntax expression, `symbol` is a unique symbol, and `<const>` is a constant-valued deterministic expression.

Semantically `assume`'s are (random) variable (generative) declarations, `observe`'s condition the distribution of `assume`'d variables by (noisily) constraining the output values of (random) functions of `assume`'d variables to match observed data, and `predict`'s are "watches" which report on (via print out) the values of variables in program traces as they are explored. In Anglican, probabilistic program interpretation is taken to be a forever-continuing exploration of the space of execution traces that obey (where hard) or reflect (where soft) the `observe`'d constraints in order to report functions (via `predict`'s) of the conditional distribution of subsets of the `assume`'d variables.

Like Scheme/Lisp, Anglican eagerly and exchangeably recursively "evaluates" subexpressions, for instance of `<expr> = (<proc> <arg> ... <arg>)`, before "applying" the procedure (which may be a random procedure or special form) resulting from evaluating the first expression `<proc>` to the value of its arguments. Anglican counts applications, reported as a computational cost proxy in Sec 4. Anglican supports several special forms, notably (`lambda (<arg> ... <arg>) <body>`) which allows creation of new procedures and (`if <pred> <cons> <alt>`) which supplies branching control flow; also `begin`, `let`, `define`, `quote`, and `cond`. Anglican exposes `eval` and `apply`. Built-in deterministic procedures include `list`, `car`, `cdr`, `cons`, `mem`, etc., and arithmetic procedures `+`, `-`, `\`, etc.

All randomness in the language originates from built-in "random primitives" of which there are two types. Elementary random primitives, such as `poisson`, `gamma`, `flip`, `discrete`, `categorical`, and `normal`, generate independent and identically distributed (i.i.d.) samples when called repeatedly with the same arguments. Exchangeable random primitives, such as `crp` or `beta-bernoulli`, return a random procedure with internal state that generates an exchangeably distributed samples when called repeatedly.

In the interpreted version of Anglican the outer `<proc>` in all `observe <expr>`'s must be a built-in random primitive, which guarantees that likelihood of output given arguments can be computed exactly.

2.2 New Clojure-Style Syntax

The compiled version of Anglican now supports a syntax that more closely integrates with the host language Clojure. The macro `defquery` defines an Anglican program from within Clojure

```
|| (defquery symbol [arg1 arg2 ...] <body>)
```

Here `symbol` is the name of the program, and the arguments may be used to pass values of parameters or observed variables to the program. The set of allowable body expressions `<body>` is a subset of the Clojure language, in which all basic Clojure language forms and first order primitives are supported. Macros and higher-order functions are not inherited from Clojure, although a subset of macros⁵ and higher-order functions⁶ has been implemented.

In the CPS version of Anglican, random primitives such as `normal` and `discrete` return first class distribution objects, rather than sampled values, and, furthermore, are user programmable without requiring modifications to the Anglican compiler. The special forms `sample` and `observe` associate values with random variables drawn from a distribution

```
|| (sample <dist>)
|| (observe <dist> <value>)
```

The language additionally provides a data type for sequences of random variables that are not i.i.d., which we refer to as a random process. A random process implements two operations

```
|| (produce <process>)
|| (absorb <process> <value>)
```

The `produce` primitive returns a distribution object for the next random variable in the sequence. The `absorb` primitive returns an updated random process instance, in which a value has been associated with the next random variable in the sequence. A random process is most commonly used to represent an exchangeably distributed sequence of random variables, but it may be used to represent any sequence of random variables for which it is possible to construct a distribution on the next variable given preceding variables. Random process constructors are customarily identified with uppercase names (e.g. `CRP`) and, in the same manner as before, are user programmable.

Finally, the `predict` form may be used at any point in the program to generate labeled output values

```
|| (predict <label> <expr>)
```

⁵when, cond

⁶map, reduce, filter, some, repeatedly, comp, partial

Given a previously defined query, inference may be performed using the `doquery` macro

```
(doquery algorithm <symbol>
  [<arg1> <arg2> ...]
  <opt1> <opt2> ...)
```

The inference algorithm may be specified using an `algorithm` keyword, and any options to the inference algorithm can be supplied as arguments as well. The `doquery` macro constructs a lazy sequence of program execution states that contain predicted values and optionally an importance weight, which may then be consumed and analyzed by, for instance, an outer Clojure or Java program.

3 Inference

An execution trace is the sequence of memory states resulting from the sequence of function applications performed during the interpretation of a program. In probabilistic programming systems like Anglican any variable may be declared as being the output of a random procedure. Such variables can take different values in independent interpretations of the program. This leads to a “many-worlds” computational trace tree in which, at interpretation time, there is a branch at every random procedure application.

To define the probability of a single execution trace, first fix an ordering of the exchangeable lines of the program and index the `observe` lines by n . Let $p(y_n|\theta_{t_n}, \mathbf{x}_n)$ be the likelihood of the `observe`’d output y_n where t_n is a random procedure type (i.e. `gamma`, `poisson`, etc.), θ_{t_n} is its argument (possibly multi-dimensional), and \mathbf{x}_n is the set of all random procedure application results computed before the likelihood of observation y_n is evaluated. Both the type t_n and the parameter θ_{t_n} can be functions of any in-scope subset of \mathbf{x}_n . We can then define the probability of an execution trace to be

$$\tilde{p}(\mathbf{y}, \mathbf{x}) \equiv \prod_{n=1}^N p(y_n|\theta_{t_n}, \mathbf{x}_n)\tilde{p}(\mathbf{x}_n|\mathbf{x}_{n-1}) \quad (1)$$

where \mathbf{y} is the set of all `observe`’d quantities, \mathbf{x} is the set of all random procedure application results, and \sim marks distributions which we can only sample.

The number and type of all random procedure applications performed before the n th `observe` may vary in one program trace to the next. We define the probability of the sequence of their outputs \mathbf{x}_n to be

$$\begin{aligned} &\tilde{p}(\mathbf{x}_n|\mathbf{x}_{n-1}) \\ &= \prod_{k=1}^{|\mathbf{x}_n \setminus \mathbf{x}_{n-1}|} p(x_{n,k}|\theta_{t_{n,k}}, x_{n,1:(k-1)}, \mathbf{x}_{n-1}). \end{aligned} \quad (2)$$

Here $\mathbf{x}_0 = \emptyset$ is the empty set and $x_{n,j:k}$ are the j to k th values generated by random procedure applications in the trace up to observation likelihood computation n . The cardinality of the set $\mathbf{x}_n \setminus \mathbf{x}_{n-1}$, notated $|\cdot|$, arises implicitly as the total number of random procedure applications in a given execution trace. As before, $\theta_{t_{n,k}}$ are the arguments to, and $t_{n,k}$ the type of, the (n, k) th random procedure – both of which may be functions of subsets of in-scope subsets of variables $x_{n,1:(k-1)} \cup \mathbf{x}_{n-1}$. Note that $\mathbf{x}_{n-1} \subseteq \mathbf{x}_n$. Also note that variable referencing defines a directed conditional dependency structure for the probability model encoded by the program, i.e. $x_{n,k}$ need not (and often cannot due to variable scoping) depend on the outputs of all previous random procedure applications.

We use sampling to explore and characterize the distribution $\tilde{p}(\mathbf{x}|\mathbf{y}) \propto \tilde{p}(\mathbf{y}, \mathbf{x})$, i.e. the distribution of all random procedure outputs that lead to different program execution traces, conditioned on observed data. Related approaches include rejection sampling [3], single-site MH [3, 12].

All members of the set of all directed probabilistic models with fixed-structure joint distributions can be expressed as probabilistic programs that “unroll” in all possible execution traces into an equivalent joint distribution. As Church-like probabilistic programming frameworks, Anglican included, support recursive procedures and branching on the values returned by random procedures, the corresponding set of models is a superset of the set of all directed graphical models. Other related efforts eschew Turing-completeness and operate on a restricted set of models [11, 9, 7, 10] where inference techniques other than sampling can more readily be employed.

3.1 A New Approach

Towards our new approach to probabilistic programming inference, first consider a standard sequential Monte Carlo (SMC) recursion for sampling from a sequence of intermediate distributions that terminates in $\tilde{p}(\mathbf{x}|\mathbf{y}) \propto \tilde{p}(\mathbf{y}, \mathbf{x})$ where \mathbf{x} and \mathbf{y} are as before, and the joint is given by Eq.’s 1 and 2. Note that a sequence of intermediate approximating distributions can be constructed from any syntactically allowed re-ordering of

$$\tilde{p}(\mathbf{x}_1)\tilde{p}(\mathbf{x}_2|\mathbf{x}_1) \cdots \tilde{p}(\mathbf{x}_n|\mathbf{x}_{n-1})p(y_1|\mathbf{x}_1) \cdots p(y_n|\mathbf{x}_n).$$

Assume that observation likelihoods are pushed as far left in this sequence of approximating distributions as possible; however it is clear how to proceed if this is not the case. Assume we have $1 \leq \ell \leq L$ unweighted samples $\mathbf{x}_{n-1}^{(\ell)} \sim \tilde{p}(\mathbf{x}_{n-1}|y_{1:(n-1)})$ and that from these we will produce approximate samples from $\tilde{p}(\mathbf{x}_n|y_{1:n})$.

To do so via importance sampling we may choose any proposal distribution $q(\mathbf{x}_n | \mathbf{x}_{n-1}, y_{1:n})$. Sampling from this and weighting by the discrepancy between it and the distribution of interest, $\tilde{p}(\mathbf{x}_n, y_{1:n})$, we arrive at samples with unnormalized weights $\tilde{w}_n^{(\ell)} = \tilde{p}(\hat{\mathbf{x}}_n^{(\ell)}, y_{1:n}) / q(\hat{\mathbf{x}}_n^{(\ell)} | \mathbf{x}_{n-1}^{(\ell)}, y_{1:n})$. Here hats notate the difference between weighted and unweighted samples, those with being weighted and vice versa.

This expression simplifies substantially in the “propose from the prior” case where the proposal distribution is defined to be the continued interpretation of the program from observation likelihood evaluation $n - 1$ to n , i.e. $q(\hat{\mathbf{x}}_n^{(\ell)} | \mathbf{x}_{n-1}^{(\ell)}, y_{1:n}) \equiv p(\hat{\mathbf{x}}_n^{(\ell)} | \mathbf{x}_{n-1}^{(\ell)})$. In this case the weight simplifies to $\tilde{w}_n^{(\ell)} = p(y_n | \hat{\mathbf{x}}_n^{(\ell)})$. Sampling an unweighted particle set $\mathbf{x}_n^{(\ell)} \sim \sum_{\ell} w_n^{(\ell)} \delta_{\hat{\mathbf{x}}_n^{(\ell)}}$, where $w_n^{(\ell)} = \tilde{w}_n^{(\ell)} / \sum_j \tilde{w}_n^{(j)}$, completely describes SMC for probabilistic program inference.

The SMC procedure described is, to first approximation, the inner loop of PMCMC. It corresponds to a procedure whereby the probabilistic program is interpreted in parallel (possibly each particle in its own thread or process) between observation likelihood calculations. Unfortunately, SMC with a finite set of particles is not itself directly viable for probabilistic programming inference for all the familiar reasons: particle degeneracy, inefficiency in models with global, continuous parameters, etc.

PMCMC, on the other hand, is directly viable. PMCMC for probabilistic programming inference is a MH algorithm for exploring the space of execution traces that uses SMC proposals internally. This, unlike prior art, allows sampling of execution traces with changes to potentially many more than one variable at a time. The particular variant of PMCMC we discuss in this paper is Particle-Gibbs (PG) although we have developed engines based on other PMCMC variants including particle independent Metropolis Hastings and conditional sequential Monte Carlo too. PG works by iteratively re-running SMC, with, on all but the first sweep, reinsertion of a “retained” particle trace into the set of particles at every stage of SMC. PG is theoretically justified as an MH transition operator that, like the Gibbs operator, always accepts [1, 4]. In this paper we describe PMCMC for probabilistic programming inference algorithmically in Alg.1 and experimentally demonstrate its relative efficacy for probabilistic programming inference.

In Alg. 1 the function $\mathbf{r}(N, \mathcal{S})$ stands for multinomial sample N items from the set of pairs $\mathcal{S} = \{\{w_1, \theta_1\}, \dots, \{w_M, \theta_M\}\}$, where each element of \mathcal{S} consists of an unnormalized weight w_m and interpreter memory states θ_m . For each sample value θ_m returned,

the function \mathbf{r} also returns the original, corresponding unnormalized weight w_m . This kind of weight book-keeping retains, for all particles, the results of the outermost `observe` likelihood function applications so that the unnormalized weights are available in the retained particle $\{w^*, \mathbf{x}^*\}$ in the next sweep.

Algorithm 1 PMCMC for Prob. Prog. Inference

```

L ← number of particles
S ← number of sweeps
{ $\tilde{w}_N^{(\ell)}, \mathbf{x}_N^{(\ell)}$ } ← Run SMC
for s < S do
  { $\cdot, \mathbf{x}_N^*$ } ←  $\mathbf{r}(1, \{1/L, \mathbf{x}_N^{(\ell)}\})$ 
  { $\cdot, \mathbf{x}_0^{(\ell)}$ } ← initialize L - 1 interpreters
  for d ∈ ordered lines of program do
    for  $\ell < L - 1$  do
       $\bar{\mathbf{x}}_{n-1}^{(\ell)} \leftarrow \mathbf{fork}(\mathbf{x}_{n-1}^{(\ell)})$ 
    end for
    if directive(d) == “assume” then
      for  $\ell < L - 1$  do
         $\bar{\mathbf{x}}_n^{(\ell)} \leftarrow \mathbf{interpret}(d, \bar{\mathbf{x}}_{n-1}^{(\ell)})$ 
      end for
      { $\mathbf{x}_n^{(\ell)}$ } ← { $\bar{\mathbf{x}}_n^{(\ell)}$ } ∪  $\mathbf{x}_n^*$ 
    else if directive(d) == “predict” then
      for  $\ell < L - 1$  do
         $\mathbf{interpret}(d, \bar{\mathbf{x}}_{n-1}^{(\ell)})$ 
      end for
       $\mathbf{interpret}(d, \mathbf{x}_{n-1}^*)$ 
    else if directive(d) == “observe” then
      for  $\ell < L - 1$  do
        { $\bar{w}_n^{(\ell)}, \bar{\mathbf{x}}_n^{(\ell)}$ } ←  $\mathbf{interpret}(d, \bar{\mathbf{x}}_{n-1}^{(\ell)})$ 
      end for
       $\mathcal{T} \leftarrow \mathbf{r}(L - 1, \{\bar{w}_n^{(\ell)}, \bar{\mathbf{x}}_n^{(\ell)}\} \cup \{\tilde{w}_n^*, \mathbf{x}_n^*\})$ 
      { $\tilde{w}_n^{(\ell)}, \mathbf{x}_n^{(\ell)}$ } ←  $\mathcal{T} \cup \{\tilde{w}_n^*, \mathbf{x}_n^*\}$ 
    end if
  end for
end for

```

In Alg. 1 “Run SMC” means running one sweep of the s loop with L particles and no retained particle, d is a program line, and `fork(·)` means to copy the entire interpreter memory datastructure (efficient implementations have characteristics similar to POSIX `fork()` [8]). The command `interpret(d, ·)` means execute line d in the given interpreter. Only when interpreting an `observe` must the interpreter return a weight, that being result of the outermost apply of the `observe` statement. Bars indicate temporary data structures, not averages. All sets are ordered with unions implemented by append operations.

Note that there are more efficient PMCMC algorithms for probabilistic programming inference. In particular, there is no reason to `fork` unless an `observe` has just

been interpreted. Alg. 1 is presented in this form for expositional purposes.

3.2 Random Database

We refer to the MH approach to sampling over the space of all traces proposed in [12] as “random database” (RDB). A RDB sampler is a MH sampler where a single variable drawn in the course of a particular interpretation of a probabilistic program is modified via a standard MH proposal, and this modification is accepted by comparing the value of the joint distribution of old and new program traces. For completeness we review RDB here, noting a subtle correction to the acceptance ratio proposed in the original reference which is proper for a larger family of models.

The RDB sampler employs a data structure that holds all random variables \mathbf{x} associated with an execution trace, along with the parameters and log probability of each draw. Note that interpretation of a program is deterministic conditioned on \mathbf{x} . A new proposal trace is initialized by picking a single variable $x_{m,j}$ from the $|\mathbf{x}|$ random draws, and resampling its value using a reversible kernel $\kappa(x'_{m,j}|x_{m,j})$. Starting from this initialization, the program is rerun to generate a new set of variables \mathbf{x}' that correspond to a new valid execution trace. In each instance where the random procedure type remains the same, we reuse the existing value from the set \mathbf{x} , rescaling its log probability conditioned on the preceding variables where necessary. When the random procedure type has changed, or a new random variable is encountered, its value is sampled in the usual manner. Finally, we compute the probability $p(\mathbf{y}|\mathbf{x}')$ by rescaling each `observe` as needed, and accept with probability

$$\min\left(1, \frac{p(\mathbf{y}|\mathbf{x}')p(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})q(\mathbf{x}'|\mathbf{x})}\right). \quad (3)$$

In order to calculate the ratio of the proposal probabilities $q(\mathbf{x}'|\mathbf{x})$ and $q(\mathbf{x}|\mathbf{x}')$, we need to account for the variables that were resampled in the course of constructing the proposal, as well as the fact that the sets \mathbf{x}' and \mathbf{x} may have different cardinalities $|\mathbf{x}'|$ and $|\mathbf{x}|$. We will use the (slightly imprecise) notation $\mathbf{x}' \setminus \mathbf{x}$ to refer to the set of variables that were resampled, and let $\mathbf{x}' \cap \mathbf{x}$ represent the set of variables common to both execution traces. The proposal probability is now given by

$$q(\mathbf{x}'|\mathbf{x}) = \frac{\kappa(x'_{m,j}|x_{m,j}) p(\mathbf{x}' \setminus \mathbf{x} | \mathbf{x}' \cap \mathbf{x})}{|\mathbf{x}| p(x'_{m,j} | \mathbf{x}' \cap \mathbf{x})}. \quad (4)$$

In our implementation, the initialization $x'_{m,j}$ is simply resampled conditioned on the preceding variables, such that $\kappa(x'_{m,j}|x_{m,j}) = p(x'_{m,j}|\mathbf{x}' \cap \mathbf{x})$. The reverse

proposal density can now be expressed in a similar fashion in terms of $\mathbf{x}' \setminus \mathbf{x}$ and $\mathbf{x} \cap \mathbf{x}' = \mathbf{x}' \cap \mathbf{x}$, allowing the full acceptance probability to be written as

$$\frac{p(\mathbf{y}|\mathbf{x}') p(\mathbf{x}' | \mathbf{x}) p(\mathbf{x} \setminus \mathbf{x}' | \mathbf{x} \cap \mathbf{x}')}{p(\mathbf{y}|\mathbf{x}) p(\mathbf{x} | \mathbf{x}') p(\mathbf{x}' \setminus \mathbf{x} | \mathbf{x}' \cap \mathbf{x})}. \quad (5)$$

4 Testing

Programming probabilistic program interpreters is a non-trivial software development effort, involving both the correct implementation of an interpreter and the correct implementation of a general purpose sampler. The methodology we employ to ensure correctness of both involves three levels of testing; 1) unit tests, 2) measure tests, and 3) conditional measure tests.

4.1 Unit and Measure Tests

In the context of probabilistic programming, unit testing includes verifying that the interpreter correctly interprets a comprehensive set of small deterministic programs. Measure testing involves interpreting short revealing programs consisting of `assume` and `predict` statements (producing a sequence of ancestral, unconditioned samples, i.e. no `observe`'s). Interpreter output is tested relative to ground truth, where ground truth is computed via exhaustive enumeration, analytic derivation, or some combination, and always in a different, well-tested independent computational system like Matlab. Various comparisons of the empirical distribution constructed from the accumulating stream of output `predict`'s and ground truth are computed; Kulback-Leibler (KL) divergences for discrete sample spaces and Kolmogorov Smirnov (KS) test statistics for continuous sample spaces. While it is possible to construct distribution equality hypothesis tests for some combinations of test statistic and program we generally are content to accept interpreters for which there is clear evidence of convergence towards zero of all test statistics for all measure tests. Anglican passed all unit and measure tests.

4.2 Conditional Measure Tests

Measure tests involving conditioning provide additional information beyond that provided by measure and unit tests. Conditioning involves endowing programs with `observe` statements which constrain or weight the set of possible execution traces. Interpreting `observe` statements engages the full inference machinery. Conditional measure test performance is measured in the same way as measure test performance. They are also how we compare different probabilistic programming inference engines.

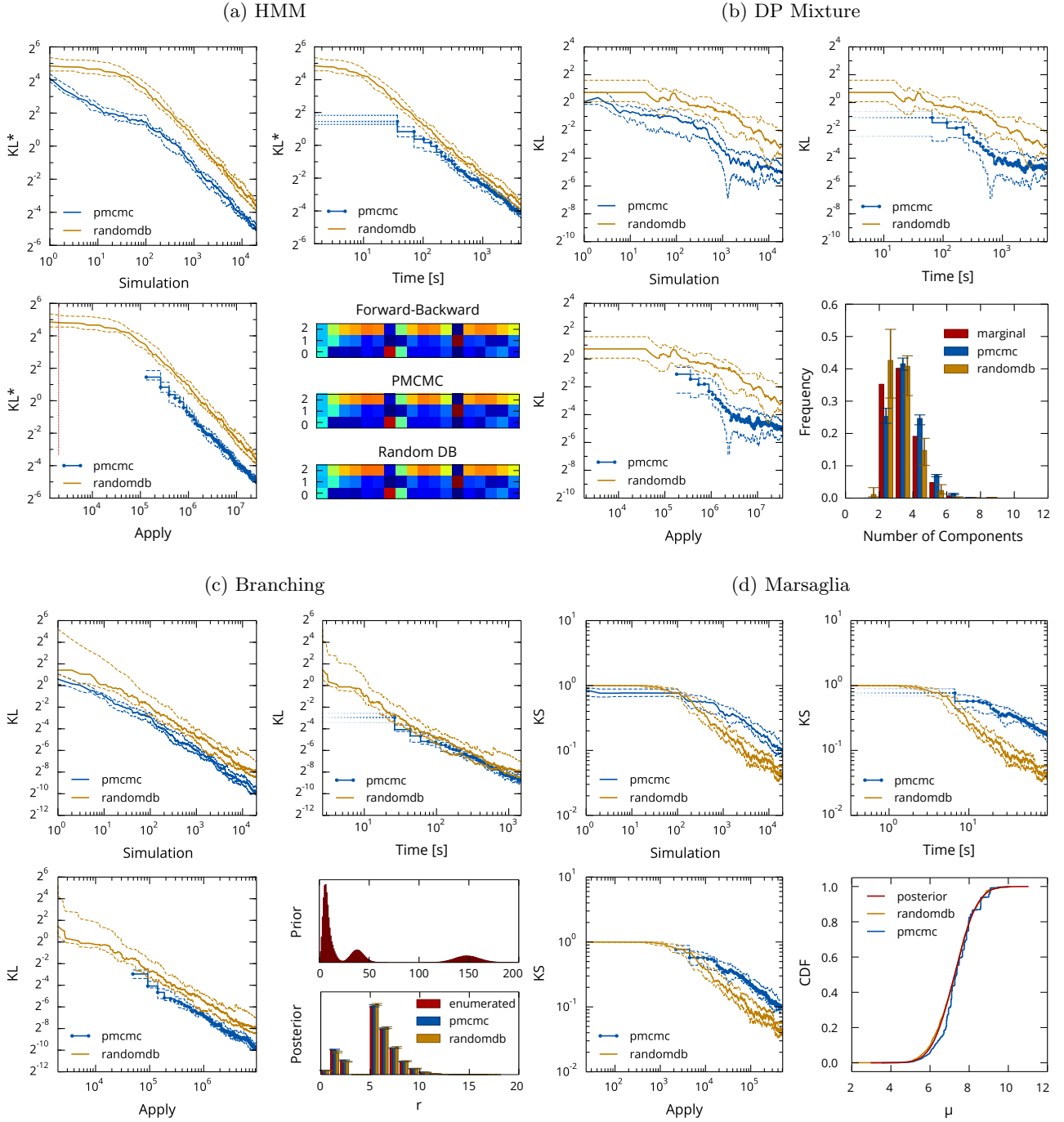


Figure 1: Comparative conditional measure test performance: PMCMC with 100 particles vs. RDB.

5 Inference Engine Comparison

We compare PMCMC to RDB measuring convergence rates for an illustrative set of conditional measure test programs. Results from four such tests are shown in Figure 1 where the same program is interpreted using both inference engines. PMCMC is found to converge faster for conditional measure test programs that cor-

respond to expressive probabilistic graphical models with rich conditional dependencies.

The four test programs are: 1) a program that corresponds to state estimation in a hidden Markov model (HMM) with continuous observations (HMM: Program 5.1, Figure 1a), 2) a program that corresponds to learning an uncollapsed Dirichlet process (DP) mix-

ture of Gaussians with fixed hyperparameters (DP Mixture: Program 5.2, Figure 1b), 3) a multimodal branching with deterministic recursion program that cannot be represented as a graphical model in which all possible execution paths can be enumerated (Branching: Program 5.3, Figure 1c), and 4) a program that corresponds to inferring the mean of a univariate normal generated via an Anglican-coded Marsaglia [6] rejection-sampling algorithm that halts with probability one and generates an unknown number of internal random variables (Marsaglia: Program 5.4, Figure 1d). We refer to 1) and 2) as “expressive” models because they have complex conditional dependency structures and 3) and 4) as simple models because the programs encode models with very few free parameters. 1) and 2) illustrate our claims; 3) and 4) are included to document the correctness and completeness of the Anglican implementation while also demonstrating that the gains illustrated in 1) and 2) do not come at too great a cost even for simple programs for which, a priori, PMCMC might be reasonably be expected to underperform.

In Figures 1a-d there are three panels that report similar style findings across test programs and a fourth that is specific to the individual test program. In all, PMCMC results are reported for a single-threaded interpreter with 100 particles. The choice of 100 particles is largely arbitrary; our results are stable for a large range of values. PMCMC is dark blue while RDB is light orange. We report the 25% (lower dashed) median (solid) and 75% (upper dashed) percentiles over 25 runs with differing random number seeds. We refer to KL/KL*/KS as distances and compute each via a running average of the empirical distribution of `predict` statement outputs to ground truth starting at the first `predict` output. Note that lower is better. We define the number of simulations to be the number of times the program is interpreted in its entirety. For RDB this means that the number of simulations is exactly the number of sampler sweeps; for PMCMC it is the number of particles multiplied by the number of sampler sweeps. The time horizontal axes report wall clock time; the apply axes report the number of function applications performed by the interpreter. In the distance vs. time plots, observed single-threaded PMCMC wall-clock times are reported via filled circles; the left-ward dotted lines illustrate hypothetically what should be achievable via parallelism. Carefully note that PMCMC requires completing a number of simulations equal to the number of particles (here 100) before emitting batched `predict` outputs. This means that single-threaded implementations of PMCMC suffer from latency that RDB does not. Still, for some programs both the quality of PMCMC’s `predict` outputs and PMCMC’s convergence rate is faster even

in direct wall clock time comparison to RDB. PMCMC appears to converge faster for some programs than RDB even relative to the number of function applications. Equivalent results were obtained relative to `eval` counts.

5.1 HMM

New

```
(defquery hmm
  [observations init-dist trans-dists obs-dists]
  (predict
    :states
    (reduce
      (fn [states obs]
        (let [state (sample (get trans-dists
                              (peek states)))]
          (observe (get obs-dists state) obs)
                (conj states state))))
      [(sample init-dist)
       observations]))
```

Original (deprecated)

```
[assume initial-state-dist (list (/ 1 3) (/ 1 3) (/ 1 3))]
[assume get-state-transition-dist (lambda (s)
  (cond ((= s 0) (list .1 .5 .4)) ((= s 1) (list .2 .2 .6))
        ((= s 2) (list .15 .15 .7))))]
[assume transition (lambda (prev-state)
  (discrete (get-state-transition-dist prev-state)))]
[assume get-state (mem (lambda (index)
  (if (<= index 0) (discrete initial-state-dist)
      (transition (get-state (- index 1))))))]
[assume get-state-observation-mean (lambda (s)
  (cond ((= s 0) -1) ((= s 1) 1) ((= s 2) 0)))]
[observe (normal (get-state-obs-mean (get-state 1)) 1) .9]
[observe (normal (get-state-obs-mean (get-state 2)) 1) .8]
:
[observe (normal (get-state-obs-mean (get-state 16)) 1) -1]
[predict (get-state 0)]
[predict (get-state 1)]
:
[predict (get-state 16)]
```

The HMM program corresponds to a latent state inference problem in an HMM with three states, one-dimensional Gaussian observations (.9, .8, .7, 0, -.025, 5, 2, 0.1, 0, .13, .45, 6, .2, .3, -1, -1), with known means and variances, transition matrix, and initial state distribution. The lines of the program were organized with the `observe`’s in “time” sequence.

The KL^* axis reports the sum of the Kulback-Leibler divergences between the running sample average state occupancy across all states of a HMM including the initial state and one trailing predictive state $KL_S^* = \sum_i D_{KL}(\frac{1}{S} \sum_{s=1}^S \delta_{z_i^{(s)}} || \gamma_i)$. Here $\delta_{z_i^{(s)}}(k)$ returns one if simulation s has latent state z at time step i equal to k and $\gamma_i(k)$ is the true marginal probability of the latent state indicator z_i taking value k at time step i . The vertical red line in the apply plot indicates the time and number of applies it takes to run forward-backward in the Anglican interpreter.

The fourth plot shows the learned posterior distribution over the latent state value for all time steps including both the initial state and a trailing predictive time step. While RDB produces a reasonable approximation to the true posterior, it does so more slowly and with greater residual error.

5.2 DP Mixture

New

```
(defquery crp-mixture
  [observations alpha mu beta a b]
  (let [precision-prior (gamma a b)]
    (loop [observations observations
          state-proc (CRP alpha)
          obs-dists {}
          states []]
      (if (empty? observations)
          (do
             (predict :states states)
             (predict :num-clusters (count obs-dists))
             (let [state (sample (produce state-proc))
                   obs-dist (get obs-dists
                                state)
                   state (let [l (sample precision-prior)
                               s (sqrt (/ (* beta 1)))
                               m (sample (normal mu s))]
                           (normal m (sqrt (/ 1)))))]
                 (observe obs-dist (first observations))
                 (recur (rest observations)
                        (absorb state-proc state)
                        (assoc obs-dists state obs-dist)
                        (conj states state)))))))))
```

Original (deprecated)

```
[assume class-generator (crp 1.72)]
[assume class (mem (lambda (n) (class-generator)))]
[assume var (mem (lambda (c) (* 10 (/ 1 (gamma 1 10)))))]
[assume mean (mem (lambda (c) (normal 0 (var c))))]
[assume u (lambda () (list (class 1) (class 2) ...
                           (class 9) (class 10)))]
[assume K (lambda () (count (unique (u))))]
[assume means (lambda (i c)
                (if (= i c) (list (mean c))
                    (cons (mean i) (means (+ i 1) c) )))]
[assume stds (lambda (i c)
              (if (= i c) (list (sqrt (* 10 (var c)))
                                (cons (var i) (stds (+ i 1) c) )))]
[observe (normal (mean (class 1)) (var (class 1))) 1.0]
[observe (normal (mean (class 2)) (var (class 2))) 1.1]
:
[observe (normal (mean (class 10)) (var (class 10))) 0]
[predict (u)]
[predict (K)]
[predict (means 1 (K))]
[predict (stds 1 (K))]
```

The DP mixture program corresponds to a clustering with unknown mean and variance problem modelled via a Dirichlet process mixture of one-dimensional Gaussians with unknown mean and variance (normal-gamma priors). The KL divergence reported is between the running sample estimate of the distribution over the number of clusters in the data and the ground truth distribution over the same. The ground truth distribution over the number of clusters was computed for this model and data by exhaustively enumerating all partitions of the data (1.0, 1.1, 1.2, -10, -15, -20, .01, .1, .05, 0), analytically computing evidence terms by exploiting conjugacy, and conditioning on partition cardinality. The fourth plot shows the posterior distribution over the number of classes in the data computed by both methods relative to the ground truth.

This program was written in a way that was intentionally antagonistic to PMCMC. The continuous likelihood parameters were not marginalized out and the `observe` statements were not organized in an optimal ordering. Despite this, PMCMC outperforms RDB per simulation, wall clock time, and apply count.

5.3 Branching

New

```
(defn fib [n]
  (loop [a 0 b 1 m 0]
    (if (= m n)
        a
        (recur b (+ a b) (inc m)))))

(with-primitive-procedures [fib]
  (defquery branching []
    (let [count-prior (poisson 4)
          r (sample count-prior)
          l (if (< 4 r)
              6
              (+ (fib (* 3 r))
                 (sample count-prior)))]
        (observe (poisson 1) 6)
        (predict :r r))))
```

Original (deprecated)

```
[assume fib (lambda (n)
              (cond ((= n 0) 1) ((= n 1) 1)
                    (else (+ (fib (- n 1)) (fib (- n 2))))))]
[assume r (poisson 4)]
[assume l (if (< 4 r) 6 (+ (fib (* 3 r)) (poisson 4)))]
[observe (poisson 1) 6]
[predict r]
```

The branching program has no corresponding graphical model. It was designed to test for correctness of inference in programs with control logic and execution paths that can vary in the number of sampled values. It also illustrates mixing in a model where, as shown in the fourth plot, there is a large mismatch between the prior and the posterior, so rejection and importance sampling are likely to be ineffective. Because there is only one observation and just a single named random variable PMCMC and RDB should and does achieve essentially indistinguishable performance normalized to simulation, time and apply count.

5.4 Marsaglia

New

```
(defm marsaglia-normal [mu std]
  (let [u (uniform-continuous -1.0 1.0)]
    (loop [x (sample u)
          y (sample u)]
      (let [s (+ (* x x) (* y y))]
        (if (< s 1.0)
            (+ mu (* std (* x (sqrt (* -2.0 (/ (log s) s))))))
            (recur (sample u) (sample u))))))

(defquery gaussian-marsaglia
  [observations sigma mu0 sigma0]
  (let [mu (marsaglia-normal mu0 sigma0)
        likelihood (normal mu sigma)]
    (reduce (fn [_ obs]
              (observe likelihood obs))
            nil
            observations)
    (predict :mu mu)))
```

Original (deprecated)

```
[assume marsaglia-normal
  (lambda (mu std)
    (define x (uniform-continuous -1.0 1.0))
    (define y (uniform-continuous -1.0 1.0))
    (define s (+ (* x x) (* y y)))
    (if (< s 1)
        (+ mu (* std (* x (sqrt (* -2.0 (/ (log s) s))))))
        (marsaglia-normal mu std)))]
[assume std (sqrt 2)]
[assume mu (marsaglia-normal 1 (sqrt 5))]
[observe (normal mu std) (+ 8 1)]
[observe (normal mu std) 8]
[predict mu]
```


Marsaglia is a test program included here for completeness. It is an example of a type of program for which PMCMC sometimes may not be more efficient. Marsaglia is the name given to the rejection form of the Box-Muller algorithm [2] for sampling from a Gaussian [6]. The Marsaglia test program corresponds to an inference problem in which observed quantities are drawn from a Gaussian with unknown mean and this unknown mean is generated by an Anglican implementation of the Marsaglia algorithm for sampling from a Gaussian. The KS axis is a Kolmogorov-Smirnov test statistic [5] computed by finding the maximum deviation between the accumulating sample and analytically derived ground truth cumulative distribution functions (CDF). Equal-cost PMCMC, RDB, and ground truth CDFs are shown in the fourth plot.

Because Marsaglia is a recursive rejection sampler it may require many recursive calls to itself. We conjecture that RDB may be faster than PMCMC here because, while PMCMC pays no statistical cost, it does pay a computational cost for exploring program traces that include many random procedure calls that lead to rejections whereas RDB, due to the implicit geometric prior on program trace length, effectively avoids paying excess computational costs deriving from unnecessarily long traces.

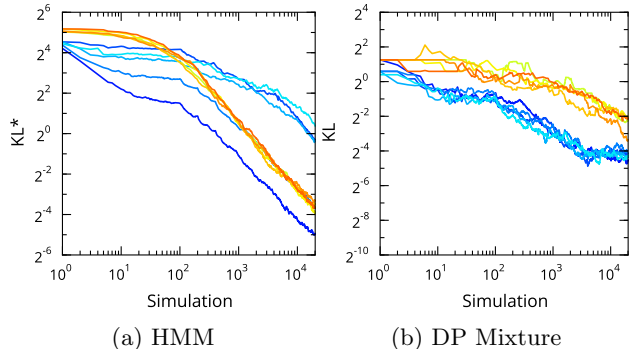


Figure 2: Effect of program line permutations

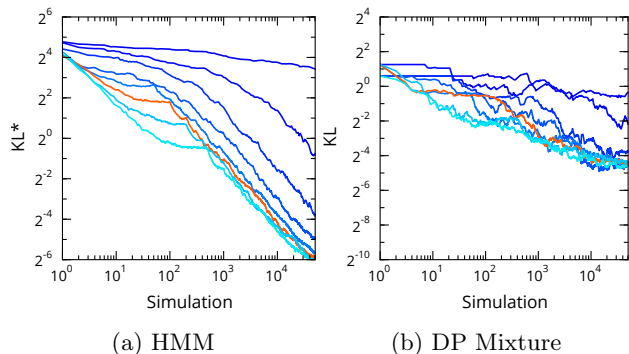


Figure 3: Effect of particle count on performance

5.5 Line Permutation

Syntactically and semantically `observe` and `predict`'s are mutually exchangeable (so too are `assume`'s up to syntactic constraints). Given this and the nature of PMCMC it is reasonable to expect that line permutations could effect the efficiency of inference. We explored this by randomly permuting the lines of the HMM and DP Mixture programs. The results are shown in Fig. 2 where blue lines correspond to median (out of twenty five) runs of PMCMC for each of twenty five program line permutations (including unmodified (dark) and reversed (light)) and orange are the same for RDB. For the HMM we found that the natural, time-sequence ordering of the lines of the program resulted in the best performance for PMCMC relative to RDB. This is because in this ordering the `observe`'s cause re-weighting to happen as soon as possible in each SMC phase of PMCMC. The effect of permuting code lines interpolates inference performance between optimal where PMCMC is best and adversarial orderings where RDB is instead. RDB performance is demonstrated here to be independent of the program line ordering.

The DP Mixture results show PMCMC outperforming RDB on all program reorderings. Further, it can be seen that the original program ordering was not optimal with respect to PMCMC inference.

While PMCMC presents the opportunity for significant gains in inference efficiency, it does not prevent programmers from seeking to further optimize performance manually. Programmers can influence inference performance by reordering program lines, in particular pushing `observe` statements as near to the front of the program as syntactically allowed, or restructuring programs to lazily rather than eagerly generate latent variables. Efficiency gains via automatic transformations or online adaptation of the ordering may be possible.

5.6 Number of Particles

Fig. 3 shows the number of particles in the PMCMC inference engine affects performance. Performance improves as a function of the number of particles. In this plot the red line indicates 100 particles. Increasing from dark to light the number of particles plotted is 2, 5, 10, 20, 50, (red, 100), 200, and 500.

6 Discussion

The PMCMC approach to probabilistic program interpretation appears to converge faster to the true conditional distribution over program execution traces

for programs that correspond to expressive models with dense conditional dependencies than single-site Metropolis-Hastings methods. That PMCMC converges faster in our tests even after normalizing for computational time is, to us, both surprising and striking. To reiterate, this includes all the computation done in all the particle executions.

Current versions of Anglican include a multi-threaded inference core but work remains to achieve optimal parallelism due, in particular, to memory organization sub-optimality leading to excessive locking overhead. Using just a single thread, PMCMC surprisingly sometimes outperforms RDB per simulation anyway in terms of wall clock time.

Our specific choice of syntactically forcing `observe`'s to be noisy requires language and interpreter level restrictions and checks. Hard constraint `observe`'s can be supported in Anglican by exposing a Dirac likelihood to programmers. Persisting in not doing so should help programmers avoid writing probabilistic programs where finding even a single satisfying execution trace is NP-hard or not-computable, but also could be perceived as requiring a non-intuitive programming style.

As explored in concurrent work on Venture, there may be opportunities to improve inference performance by partitioning program variables, either automatically or via syntax constructs, and treating them differently during inference. Then, for instance, some could be sampled via plain MH, some via conditional SMC. One simple way to do this would be to combine conditional SMC and RDB MH.

Acknowledgments

We thank Xerox and Google for their generous support. We also thank Arnaud Doucet, Brooks Paige, and Yura Perov for helpful discussions about PMCMC and probabilistic programming in general.

References

- [1] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.
- [2] George EP Box and Mervin E Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.
- [3] Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Daniel Tarlow. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.
- [4] Roman Holenstein. *Particle Markov Chain Monte Carlo*. PhD thesis, The University of British Columbia, 2009.
- [5] Hubert W Lilliefors. On the Kolmogorov-Smirnov test for normality with mean and variance unknown. *Journal of the American Statistical Association*, 62(318):399–402, 1967.
- [6] George Marsaglia and Thomas A Bray. A convenient method for generating normal variables. *Siam Review*, 6(3):260–264, 1964.
- [7] Tom Minka, J Winn, J Guiver, and D Knowles. Infer.NET 2.4, 2010. Microsoft Research Cambridge.
- [8] The Open Group. IEEE Std 1003.1, 2004 Edition, 2004. URL <http://pubs.opengroup.org/onlinepubs/009695399/functions/fork.html>.
- [9] Avi Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI*, pages 733–740. Citeseer, 2001.
- [10] David Spiegelhalter, Andrew Thomas, Nicky Best, and Wally Gilks. Bugs 0.5: Bayesian inference using Gibbs sampling manual (version ii). *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, 1996.
- [11] Stan Development Team. Stan modeling language user's guide and reference manual. <http://mc-stan.org/>, 2013.
- [12] David Wingate, Andreas Stuhlmüller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, page 131, 2011.